StarRocks Runtime Filter 源码解析

一. 动机

RuntimeFilter是提升StarRocks执行引擎性能的关键feature之一, RuntimeFilter中的Runtime, 顾名思义, 是指Query生成物理执行计划后, 在执行引擎中evaluation时, 动态构建的Filter, 区别于优化器预先规划的Filter, 因此在有的系统中, 也被称作DynamicFilter. 当Query执行时, HashJoin的右表构建Hash表, 待Hash表构建完成后, 利用Hash表生成RuntimeFilter, 然后HashJoin左侧算子(比如ExchangeNode, OlapScanNode)使用RuntimeFilter过滤数据, 减少行数, 从而降低磁盘IO, 网络IO和算子工作量, 最使Query的计算性能显著提高.

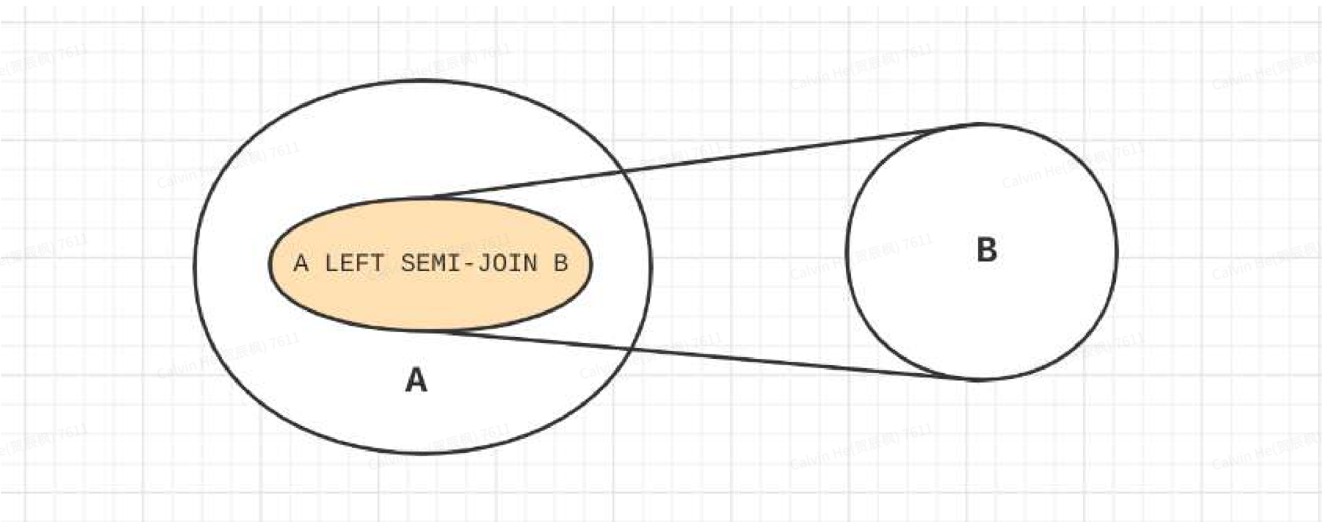
本文将详细讲述StarRocks的RuntimeFilter相关知识, 阅读完本文之后, 读者将获悉RuntimeFilter的一般知识, 复用RuntimeFilter的设计思路, 实现其他比较复杂的逻辑.

二. 背景知识

2.1. RuntimeFilter的简单介绍

在HashJoin中, 对于INNER JOIN, 下面关系代数恒等式成立:

SQL
```sql
A JOIN B = (A LEFT SEMI-JOIN B) JOIN B
```



如上图所示, 待B表建完HT后, 把HT的key Set发给A表, A表筛选出可命中key Set的tuple集合, 即A LEFT SEMI-JOIN B, 使用已选出tuple集合来probe HT. 这样做的优势有:

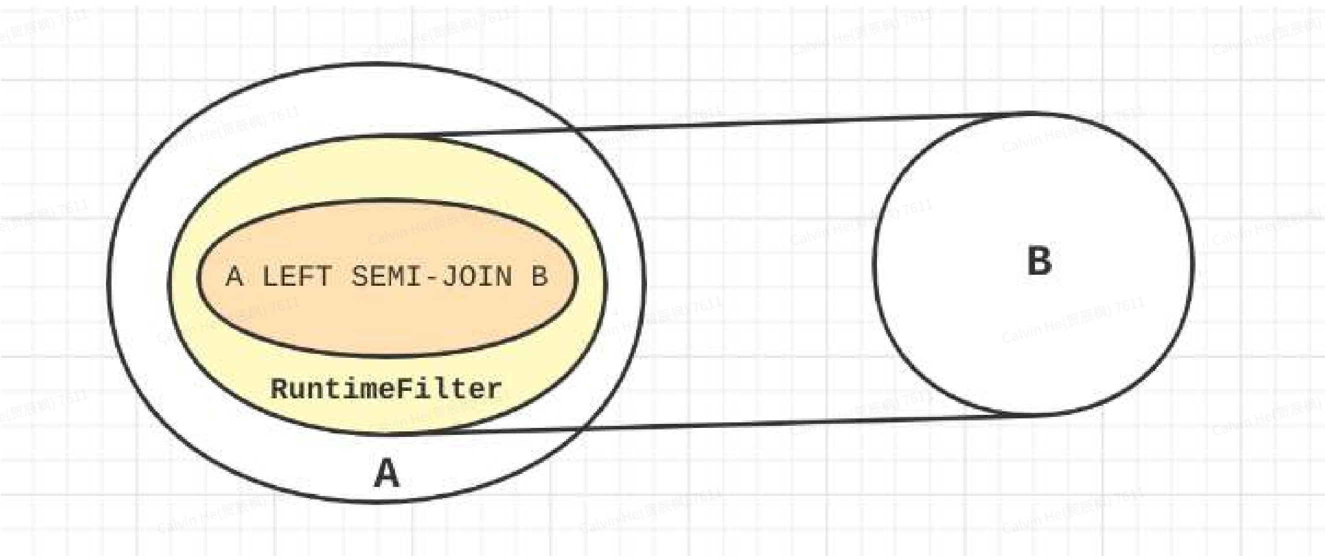
1. 降低Probe HT的输入行数;
2. 将过滤下推到存储引擎层, 可降低IO次数;

3. 如果A表的数据需要跨越机器, 在网络发送之前减少行数, 可以减少网络传输.
4. HashJoin左侧其他算子需要处理的行数减少.

事实上需要考虑的HT的keySet的规模, 避免key Set过大而引入的开销, 而引入了RuntimeFilter机制, 在StarRocks中, RuntimeFilter的类型有:

1. 0<HT.keySet().size()<=1024, 将keySet转换为in filter.
2. 0<HT.keySet().size()<=1024000, 将keySet转换为bloom filter.
3. Min-max filter, 生成bloom filter同时, 生成Min-max filter.

如下图所示, RuntimeFilter选出的数据需要包含A LEFT SEMI-JOIN B, 其中in-filter没有假阳性, bloom filter和min-max都有假阳性, RuntimeFilter尽最大可能削减IO次数, Exchange算子网络传输数据量和其他算子的处理的行数.

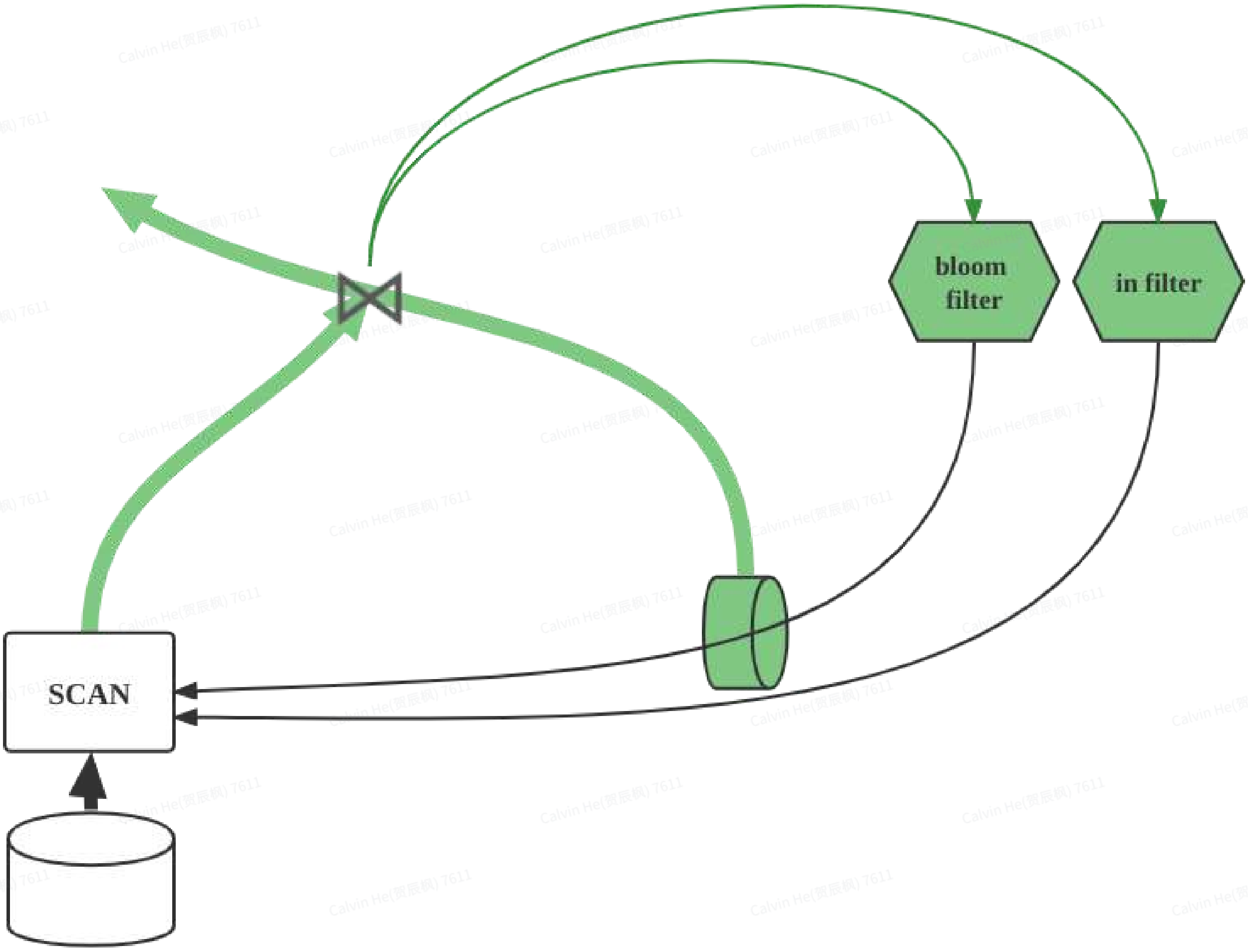


JOIN的类型比较多, 能够使上文中含LEFT SEMI-JOIN恒等式成立的JOIN, 都可以构建RF过滤HashJoin左侧的数据:

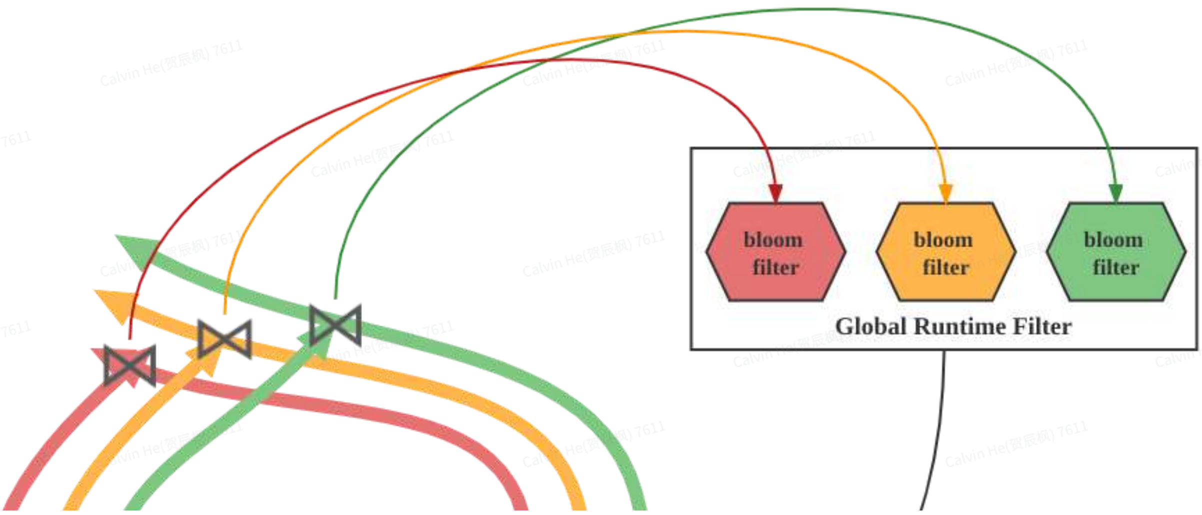
1. 可以使用RF的类型: INNER JOIN, RIGHT OUTER JOIN, LEFT SEMI JOIN, RIGHT SEMI JOIN, RIGHT ANTI JOIN.
2. 无法使用RF的类型: FULL JOIN, LEFT OUTER JOIN, LEFT ANTI JOIN.
3. LEFT ANTI JOIN其实可以使用not in filter过滤, 但是考虑到其他谓词过滤有假阳性问题, 使用RF不安全.

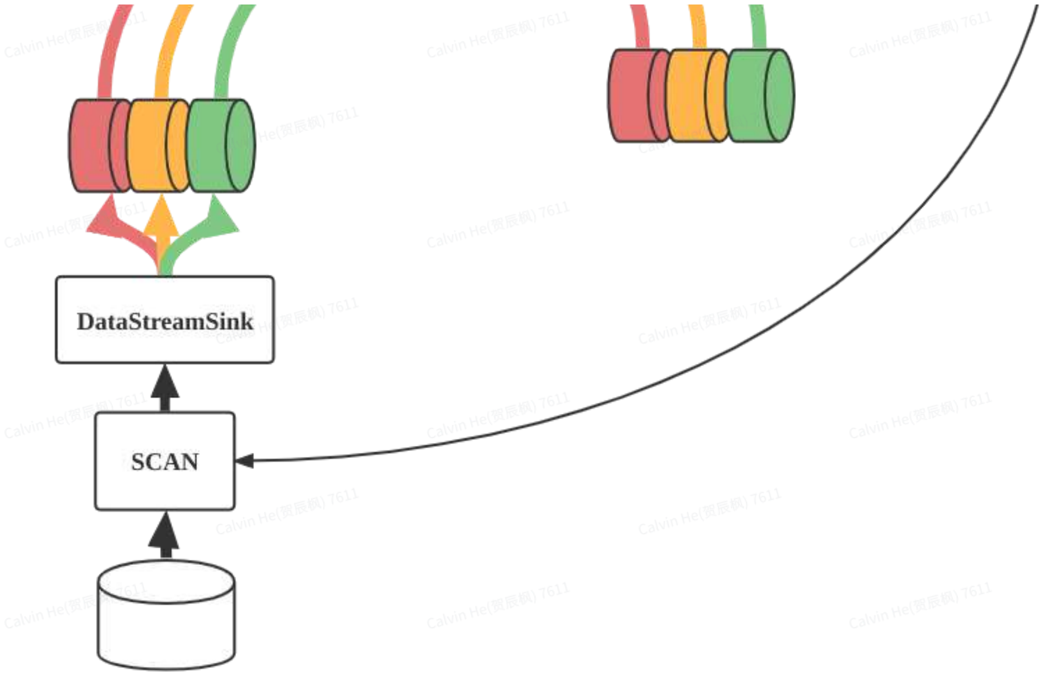
2.2. Local RuntimeFilter和Global RuntimeFilter

Local RuntimeFilter是产生和消费RuntimeFilter的算子都处于一个Fragment Instance内, 目前Local RuntimeFilter的类型有: in-filter和bloom-filter(max-min filter).



Global RuntimeFilter是指产生和消费RuntimeFilter的算子是可以跨多个Fragment Instance, 需要GRF coordinator参与GRF的分量收集, 合并和投递.





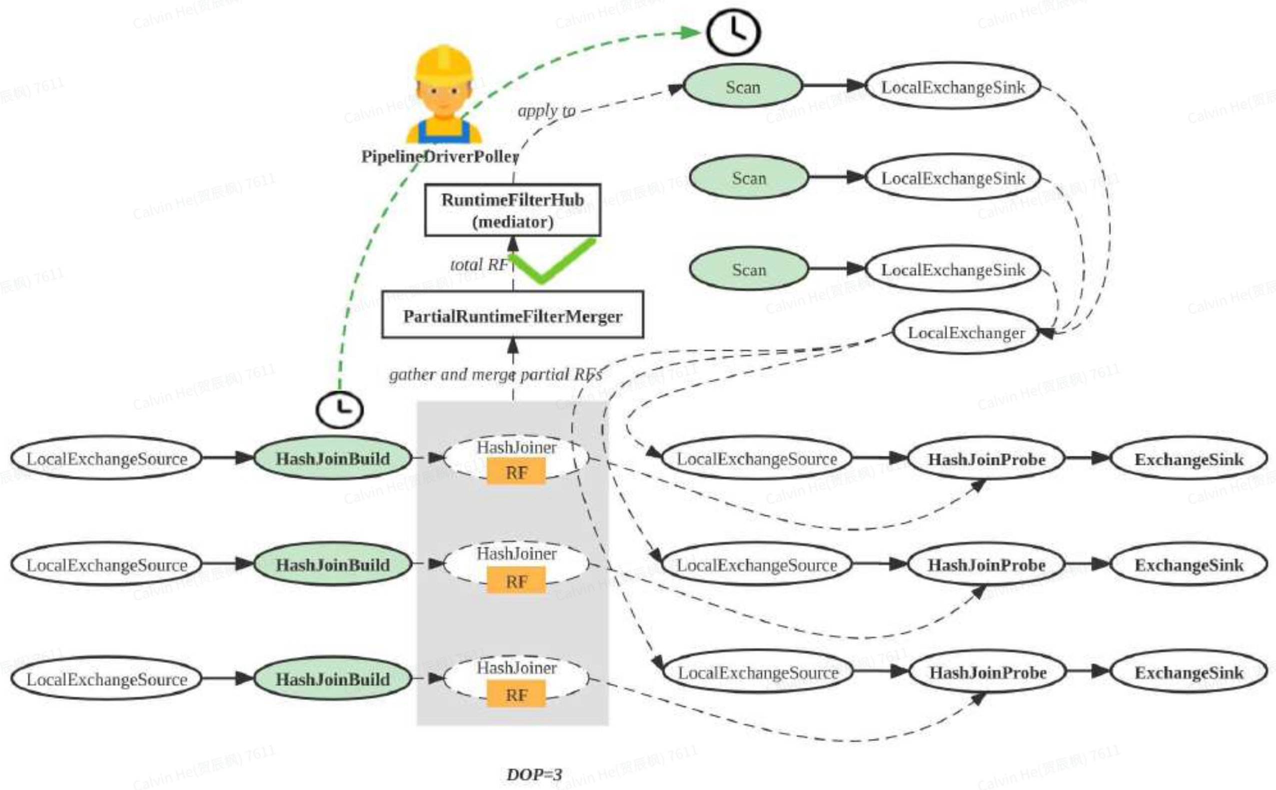
Global RuntimeFilter会在Shuffle Join中构建， Shuffle Join会把左表和右表数据分成N个对应的Partition, 每个HashJoin算子处理其中一路, 如下图所示, 图中分成了三路，分别用颜色红黄绿表示, 有三个HashJoin实例并行执行, 而GRF也有三个分量, 每个分量来自HashJoin实例的右表构建的Hash表. 这三个分量会在GRF Coordinator接的上, 拼装成一个完成GRF, 然后发送给左侧的Scan算子, 用于过滤数据.

GRF只有bloom-filter类型, 是因为右表比较大时并且HashJoin的方式是shuffle join时, 构建in-filter需要的代价比较高, 而bloom-filter的代价较小, bloom-filter需要在字节数约等于Hash表size.

2.3. Pipeline引擎中RuntimeFilter的合并问题

在Pipeline引擎中, HashJoin的一个实例, 会进一步地拆分成多个PipelineDriver, 每个PipelineDriver构建自己的Hash表, 从这些Hash表产生的RuntimeFilter是原来Local RF和Global RF分量的分量, 需

要做合并.



如上图所示, 原来一个Fragment Instance的内HashJoin实例, 根据pipeline_dop, 拆成了3路并行, HashJoinBuildOperator建完Hash表后，在HashJoiner中构建RuntimeFilter, 此时的单独的RuntimeFilter还不能在HashJoinProbeOperator一侧的下游ScanOperator上使用, 需要PartialRuntimeFilterMerger收集到所有分量之后, 才能合并出一个Local RF或者Global RF分量.

三. 源码解析

3.1. 基本概念

RuntimeFilterWorker

参考源码文件: be/src/runtime/runtime_filter_worker.cpp

RuntimeFilterWorker是合并GRF的组件, 每个BE只有一个实例, 对象实例是ExecEnv::_runtime_filter_worker; RuntimeFilterWorker启动一个内部线程, 不断地从内部的事件队列获取事件, 进行处理; 事件类型有:

- OPEN_QUERY: 用于注册RuntimeFilterMerger, RuntimeFilterMerger处理一个query的所有GRF合并和合并后total GRF传递. 执行Query的root Fragment Instance(含ResultSink)的BE上RuntimeFilterWorker是GRF Coordinator, 初始化root Fragment Instance时, 调用RuntimeFilterWorker::open_query函数注册RuntimeFilterMerger.
- SEND_PART_RF: 用于发送GRF分量, GRF分量由生产者所在的BE发给GRF Coodinator.

- RECEIVE_PART_RF: 用于接受GRF分量, 收到所有GRF分量后, 使用RuntimeFilterMerger合并成一个total GRF, 然后将total GRF发送给目标Fragment Instance.
- RECEIVE_TOTAL_RF: 用于接收total RF, 目标Fragment Instance收到total RF, 调用RuntimeFilterPort::receive_shared_runtime_filter函数, 将JoinRuntimeFilter填充到目标RuntimeFilterProbeDescriptor中.
- CLOSE_QUERY: 用于Query执行结束后移除RuntimeFilterMerger.

也就是说, RuntimeFilterWorker是GRF合并工作的participant, 其中只有一个注册了RuntimeFilterMerger的RuntimeFilterWorker是GRF coordinator. 协调者需要处理OPEN_QUERY, RECEIVE_PART_RF, CLOSE_QUERY事件, 参与者需要处理SEND_PART_RF和RECEIVE_TOTAL_RF事件. 那个BE做GRF coordinator, 是不固定的, 随query变化, 每个Query有自己唯一的RuntimeFilterMerger, Query的root Fragment Instance所在RuntimeFilterWorker是该Query的GRF coordinator.

RuntimeFilterWorker主要和RuntimeFilterPort打交道.

RuntimeFilterPort

每个Fragment Instance有自己私有RuntimeFilterPort, 对象实例为RuntimeState::_runtime_filter_port. RuntimeFilterPort的职责有:

- 使用函数add_listener注册RuntimeFilterProbeDescriptor.

```
add_listener
└── ExecNode::register_runtime_filter_descriptor      [vim be/src/exec/exec_node.cpp +150]
    ├── ExecNode::init_join_runtime_filters    [vim be/src/exec/exec_node.cpp +156]
    └── HashJoinNode::_create_implicit_local_join_runtime_filters      [vim be/src/exec/vectorized/hash_join_node.cpp +914]
```

- RuntimeFilterWorker调用RuntimeFilterPort::receive_shared_runtime_filter函数安装收到的total GRF. Total GRF被填写到RuntimeFilterProbeDescriptor::_shared_runtime_filter字段中.

```
receive_shared_runtime_filter
├── FragmentMgr::receive_runtime_filter       [vim be/src/runtime/fragment_mgr.cpp +477]
│   └── RuntimeFilterWorker::_receive_total_runtime_filter      [vim be/src/runtime/runtime_filter_worker.cpp +457]
└── receive_total_runtime_filter_pipeline       [vim be/src/runtime/runtime_filter_worker.cpp +429]
    └── RuntimeFilterWorker::_receive_total_runtime_filter      [vim be/src/runtime/runtime_filter_worker.cpp +457]
```

- HashJoin算子调用RuntimeFilterPort::publish_runtime_filters函数发布GRF分量, 该函数首先会将GRF分量填写到RuntimeFilterProbeDescriptor::_runtime_filter字段中, 然后如果对应GRF有remote targets, 则调用RuntimeFilterWorker::send_part_runtime_filter函数发送.

RuntimeFilterMerger

每个Query执行期间, 拥有一个RuntimeFilterMerger, 位于GRF coordintor中. 随Query的生命周期创建和销毁. RuntimeFilterMerger主要的职责是管理处理所属Query的GRF合并任务.

RuntimeFilterMerger::merge_runtime_filter函数, 收到最后一个GRF分量, 调用_send_total_runtime_filter函数, 合并和发送total GRF.

RuntimeFilterBuildDescriptor

HashJoin算子(非pipeline引擎，是HashJoinNode, pipeline执行引擎是HashJoinBuildOperator)上, 为每个GRF创建一个RuntimeFilterBuildDescriptor, 用于构建Hash表后, 基于Hash表构建Runtime bloom-filter.

RuntimeFilterProbeDescriptor

RuntimeFilterProbeDescriptor是算子中真正用来过滤数据GRF实例, RuntimeFilterProbeDescriptor::_runtime_filter字段保存当前Fragment Instance内GRF分量, 因为该分量可以在Fragment Instance内过滤数据, 而_shared_runtime_filter字段保存total GRF, total GRF可以跨Fragment Instance过滤数据.

算子中有一个RuntimeFilterProbeCollector对象管理属于自己的RuntimeFilterProbeDescriptor, 在非Pipeline引擎中, 是ExecNode::_runtime_filter_collector; 在Pipeline执行引擎中是OperatorFactory::

_runtime_filter_collector, Operator实例共享OperatorFactory中RuntimeFilterProbeCollector.

pipeline引擎中,算子调用Operator::eval_runtime_bloom_filters函数过滤数据.

PHP

```
void Operator::eval_runtime_bloom_filters(vectorized::Chunk* chunk) {
    if (chunk == nullptr || chunk->is_empty()) {
        return;
    }

    if (auto* bloom_filters = runtime_bloom_filters()) {
        _init_rf_counters(true);
        bloom_filters->evaluate(chunk, _bloom_filter_eval_context);
    }

    ExecNode::eval_filter_null_values(chunk, filter_null_value_columns());
}
```

```
eval_runtime_bloom_filters
├── AggregateBlockingSourceOperator::pull_chunk       [vim be/src/exec/pipeline/aggregate/aggregate_blocking_source_operator.cpp +26]
├── AggregateStreamingSourceOperator::pull_chunk      [vim be/src/exec/pipeline/aggregate/aggregate_streaming_source_operator.cpp +59]
├── AggregateDistinctStreamingSourceOperator::pull_chunk      [vim be/src/exec/pipeline/aggregate/aggregate_distinct_streaming_source_operator.cpp +41]
├── AggregateDistinctBlockingSourceOperator::pull_chunk       [vim be/src/exec/pipeline/aggregate/aggregate_distinct_blocking_source_operator.cpp +26]
├── ExchangeSourceOperator::pull_chunk      [vim be/src/exec/pipeline/exchange/exchange_source_operator.cpp +33]
├── ExchangeMergeSortSourceOperator::pull_chunk       [vim be/src/exec/pipeline/exchange/exchange_merge_sort_source_operator.cpp +47]
├── ProjectOperator::push_chunk      [vim be/src/exec/pipeline/project_operator.cpp +25]
└── OlapScanOperator::pull_chunk      [vim be/src/exec/pipeline/olap_scan_operator.cpp +127]
```

非pipeline引擎中, 算子调用ExecNode::eval_join_runtime_filters函数过滤数据.

JavaScript

```
void ExecNode::eval_join_runtime_filters(vectorized::Chunk* chunk) {
    if (chunk == nullptr) return;
    _runtime_filter_collector.evaluate(chunk);
    eval_filter_null_values(chunk);
}
```

```
eval_join_runtime_filters
├── ExecNode::eval_join_runtime_filters        [vim be/src/exec/exec_node.cpp +653]
├── ExchangeNode::get_next     [vim be/src/exec/exchange_node.cpp +115]
├── DistinctBlockingNode::get_next     [vim be/src/exec/vectorized/aggregate/distinct_blocking_node.cpp +98]
├── HdfsScanNode::get_next     [vim be/src/exec/vectorized/hdfs_scan_node.cpp +501]
├── AggregateStreamingNode::get_next   [vim be/src/exec/vectorized/aggregate/aggregate_streaming_node.cpp +29]
├── DistinctStreamingNode::get_next    [vim be/src/exec/vectorized/aggregate/distinct_streaming_node.cpp +31]
├── AggregateBlockingNode::get_next    [vim be/src/exec/vectorized/aggregate/aggregate_blocking_node.cpp +125]
├── HashJoinNode::_probe       [vim be/src/exec/vectorized/hash_join_node.cpp +569]
├── HashJoinNode::_probe_remain        [vim be/src/exec/vectorized/hash_join_node.cpp +689]
├── OlapScanNode::get_next     [vim be/src/exec/vectorized/olap_scan_node.cpp +86]
├── RepeatNode::get_next       [vim be/src/exec/vectorized/repeat_node.cpp +100]
└── ProjectNode::get_next      [vim be/src/exec/vectorized/project_node.cpp +110]
```

使用Runtime bloom filter过滤数据时, 会根据每个filter的selectivity, 自适应选择过滤性好的filter过滤, 降低过滤数据的cost.

RuntimeFilterHub

Pipeline执行引擎中使用RuntimeFilterHub管理Runtime in-filter, 在原来非Pipeline执行引擎中, 采用下推逻辑, 将构建后in-filter, 一直往当前算子的孩子算子下推; 但是在Pipeline引擎中, 这种树形结构遭到破坏, 无法采用类似的下推逻辑. 具体哪些算子使用in-filter, 事先由FE规划好, 如此以来, 产生in-filter的HashJoinBuildOperator算子RuntimeFilterHub::set_collector函数安装in-filter, 而消费in-filter的算子根据FE事先规划好的rf_waiting_set, 调用RuntimeFilterHub::gather_holders函数获得in-filter. 即RuntimeFilterHub是runtime in-filter的生产者和消费者之间的mediator. RuntimeFilterHub也是Fragment Instance私有的对象实例.

PartialRuntimeFilterMerger

PartialRuntimeFilterMerger只用于Pipeline引擎, 多个HashJoinBuildOperator算子产生的driver级的RF, 通过add_partial_filters函数合并. 来自同一个Fragment Instance的HashJoinNode产生多个HashJoinBuildOperator算子共享一个PartialRuntimeFilterMerger.

3.1. 产生和合并RuntimeFilter分量

RuntimeFilter是基于HashJoin的Hash表构建的, 我们需要查看HashJoin算子。

在Pipeline执行引擎中, HashJoinBuildOperator::set_finish构建RuntimeFilter:

Rust

```cpp
//file: be/src/exec/pipeline/hashjoin/hash_join_build_operator.cpp

void HashJoinBuildOperator::set_finishing(RuntimeState* state) {
    _is_finished = true;
    _hash_joiner->build_ht(state);

    size_t merger_index = _driver_sequence;
    if (_distribution_mode == TJoinDistributionMode::BROADCAST) {
        // As for BROADCAST, only the first finished builder creates runtime filters.
        bool expected = false;
        if (!_any_broadcast_builder_finished.compare_exchange_strong(expected, true)) {
            _hash_joiner->enter_probe_phase();
            return;
        }

        merger_index = 0;
    }

    _hash_joiner->create_runtime_filters(state);

    auto ht_row_count = _hash_joiner->get_ht_row_count();
    auto& partial_in_filters = _hash_joiner->get_runtime_in_filters();
    auto& partial_bloom_filter_build_params = _hash_joiner->get_runtime_bloom_filter_build_params();
    auto& partial_bloom_filters = _hash_joiner->get_runtime_bloom_filters();
    // add partial filters generated by this HashJoinBuildOperator to PartialRuntimeFilterMerger to merge into a
    // total one.
    auto status = _partial_rf_merger->add_partial_filters(merger_index, ht_row_count, std::move(partial_in_filters),
                                                          std::move(partial_bloom_filter_build_params),
                                                          std::move(partial_bloom_filters));
    if (status.ok() && status.value()) {
        auto&& in_filters = _partial_rf_merger->get_total_in_filters();
        auto&& bloom_filters = _partial_rf_merger->get_total_bloom_filters();

        // publish runtime bloom-filters
        state->runtime_filter_port()->publish_runtime_filters(bloom_filters);
        // move runtime filters into RuntimeFilterHub.
        runtime_filter_hub()->set_collector(_plan_node_id, std::make_unique<RuntimeFilterCollector>(
                                                                   std::move(in_filters), std::move(bloom_filters)));
```

```
    }

    _hash_joiner->enter_probe_phase();
}
```

- HashJoiner::build_ht函数构建Hash表.
- HashJoiner::create_runtime_filters函数构建RuntimeFilter.
- HashJoiner::get_runtime_in_filters函数获得以构建好的partial runtime in-filter.
- HashJoiner::get_runtime_bloom_filter_build_params函数获得partial bloom-filter构造所需的参数.
- HashJoiner::get_runtime_bloom_filters函数获得存储bloom filter的对象RuntimeFilterBuildDescriptor.
- PartialRuntimeFilterMerger::add_partial_filters负责收集partial runtime filter, 如果HashJoinBuildOperator算子所在的pipeline的并行度为N, 则需要收集N个partial runtime filter, 收集到最后一个, add_partial_filters函数把所有的partial filter合并成完整的runtime filter.
- RuntimeFilterPort::publish_runtime_filters函数负责runtime bloom filter的分发, 如果该bloom filter为local filter, 则直接调用RuntimeFilterPort::receive_runtime_filter完成本地的投递. 如果为global RF, 则此时的runtime filter只是GRF的分量, 需要调用RuntimeFilterWorker::send_part_runtime_filter函数向GRF coordinator发送分量, RuntimeFilterWorker是参与Global RF合并的全局对象, 该对象内部有一个线程, 负责发送, 接受partial GRF, 将全部收到的partial GRF合并成total GRF, 然后向目标Fragment Instance中的目标算子发送total GRF.

```
send_part_runtime_filter
└─ RuntimeFilterPort::publish_runtime_filters        [vim be/src/runtime/runtime_filter_worker.cpp +50]
   ├─ HashJoinBuildOperator::set_finishing  [vim be/src/exec/pipeline/hashjoin/hash_join_build_operator.cpp +43]
   └─ HashJoinNode::_do_publish_runtime_filters     [vim be/src/exec/vectorized/hash_join_node.cpp +885]
```

- RuntimeFilterHub::set_collector函数负责将in-filter和bloom filter添加到RuntimeFilterHub中, 这样做的目的有两个:
    - 第一, 消费in-filter和bloom-filter的PipelineDriver共享一份对象, RuntimeFilter作为mediator, 持有RuntimeFilter.
    - 第二, Pipeline执行引擎中, 产生RuntimeFilter的PipelineDriver和消费RuntimeFilter的PipelineDriver之间是异步调用的, 但是为了保证消费方在生产方完成之后执行, FE会事先规划好消费方需要等待的RuntimeFilter集合rf_wait_set, 在PipelineDriver调度时, 如果PipelineDriver有需要等待的RuntimeFilter集合, 则PipelineDriver会设置成PRECONDITION_BLOCK状态, 放入阻塞队列, 有轮询线程查看rf_waiting_set中RuntimeFilter在RuntimeFilterHub中是否已经就绪, 如果全部就绪, 说明PipelineDriver需要等待的所有RuntimeFilter已经通过set_collector函数添加完成, 此时PipelineDriver从PRECONDITION_BLOCK转化为READY状态, 就可以使用RuntimeFilter了.

3.2. Local RuntimeFilter下推逻辑

Local Runtime Filter有两种, 一种是bloom-filter, 一种是in-filter.

bloom-filter下推逻辑比较简单, 因为bloom-filter在那个算子上生效, FE事先规划好的.

In-filter 下推逻辑在非pipeline引擎中, 也比较简单, 递归调用ExecNode::push_down_predicate函数, 从HashJoin的probe一侧的算子开始, 反复下推即可. 在Pipeline执行引擎中, 首先每个算子有一个local——rf_waiting_set.

FE实现为ExecNode规划好local_rf_waiting_set, 该计划告诉ExecNode, 需要等待那些HashJoin算子产生的Runtime in-filter.

PHP

```
Status ExecNode::init(const TPlanNode& tnode, RuntimeState* state) {
    VLOG(2) << "ExecNode init:\n" << apache::thrift::ThriftDebugString(tnode);
    _runtime_state = state;
    RETURN_IF_ERROR(Expr::create_expr_trees(_pool, tnode.conjuncts, &_conjunct_ctxs));
    RETURN_IF_ERROR(init_join_runtime_filters(tnode, state));
    if (tnode.__isset.local_rf_waiting_set) {
        _local_rf_waiting_set = tnode.local_rf_waiting_set;
    }
    return Status::OK();
}
```

非pipeline算子拆解成Pipeline算子时, 调用ExecNode::init_runtime_filter_for_operator函数将local_rf_waiting_set传送给Pipeline算子.

PipelineDriver在prepare阶段, 收集所有的算子的local_rf_waiting_set, 获得all_local_rf_set, 然后在RuntimeFilterPort中设置结束Runtime in-filter的Holder对象.

C++

```cpp
// PipelineDriver::prepare
LocalRFWaitingSet all_local_rf_set;
for (auto& op : _operators) {
    if (auto* op_with_dep = dynamic_cast<DriverDependencyPtr>(op.get())) {
        _dependencies.push_back(op_with_dep);
    }

    const auto& rf_set = op->rf_waiting_set();
    all_local_rf_set.insert(rf_set.begin(), rf_set.end());

    const auto* global_rf_collector = op->runtime_bloom_filters();
    if (global_rf_collector != nullptr) {
        for (const auto& [_, desc] : global_rf_collector->descriptors()) {
            _global_rf_descriptors.emplace_back(desc);
        }

        _global_rf_wait_timeout_ns =
                std::max(_global_rf_wait_timeout_ns, global_rf_collector->wait_timeout_ms() * 1000L * 1000L);
    }
}
_local_rf_waiting_set_counter->set((int64_t)all_local_rf_set.size());
_local_rf_holders = fragment_ctx()->runtime_filter_hub()->gather_holders(all_local_rf_set)
```

需要等待RuntimeFilter的PipelineDriver状态为PRECONDITION_BLOCK, 会先放入到PipelineDriverPoller的阻塞队列中

PHP

```php

void GlobalDriverDispatcher::dispatch(DriverRawPtr driver) {
    if (driver->is_precondition_block()) {
        driver->set_driver_state(DriverState::PRECONDITION_BLOCK);
        driver->mark_precondition_not_ready();
        this->_blocked_driver_poller->add_blocked_driver(driver);
    } else {
        driver->dispatch_operators();

        // Try to add the driver to poller first.
        if (!driver->source_operator()->is_finished() && !driver->source_operator()->has_output()) {
            driver->set_driver_state(DriverState::INPUT_EMPTY);
            this->_blocked_driver_poller->add_blocked_driver(driver);
        } else {
            this->_driver_queue->put_back(driver);
        }
    }
}
```

PipelineDriverPoller会反复调用PipelineDriver::is_precondtion_block函数检查所有的runtime in-filter是否就绪.

JavaScript

```
bool is_precondition_block() {
    if (!_wait_global_rf_ready) {
        if (dependencies_block() || local_rf_block()) {
            return true;
        }
        _wait_global_rf_ready = true;
        if (_global_rf_descriptors.empty()) {
            return false;
        }
        // wait global rf to be ready for at most _global_rf_wait_time_out_ns after
        // both dependencies_block and local_rf_block return false.
        _global_rf_wait_timeout_ns += _precondition_block_timer_sw->elapsed_time();
        return global_rf_block();
    } else {
        return global_rf_block();
    }
}
```

is_precondition_block函数, 返回false表示PipelineDriver已经就绪了, 就绪条件是:

- 该算子依赖的PipelineDriver已经完成, dependencies_block返回true;
- 且依赖的local rf已经就绪.
- 且global RF已经完成或者最多等待20ms.

3.3. RuntimeFilter的下推存储引擎

RuntimeFilter in-filter下推存储层的逻辑和普通的in-filter下推逻辑相同, 主要是OlapScanConjunctsManager::normalize_predicate函数调用

OlapScanConjunctsManager::normalize_in_or_equal_predicate选择下推存储的in-filter.

OlapScanConjunctsManager::normalize_predicate调用

normalize_join_runtime_filter函数将Runtime bloom-filter中的max-min下推到存储层.

C++

```
template <PrimitiveType SlotType, typename RangeValueType>
void OlapScanConjunctsManager::normalize_join_runtime_filter(const SlotDescriptor& slot,

ColumnValueRange<RangeValueType>* range) {
    // in runtime filter
```

```
    // in runtime filter
    const auto& conjunct_ctxs = (*conjunct_ctxs_ptr);

    for (size_t i = 0; i < conjunct_ctxs.size(); i++) {
        if (normalized_conjuncts[i]) {
            continue;
        }

        const Expr* root_expr = conjunct_ctxs[i]->root();
        if (TExprOpcode::FILTER_IN == root_expr->op()) {
            const Expr* l = root_expr->get_child(0);
            if ((l->node_type() != TExprNodeType::SLOT_REF) ||
                (l->type().type != slot.type().type && !ignore_cast(slot, *l))) {
                continue;
            }
            std::vector<SlotId> slot_ids;
            if (1 == l->get_slot_ids(&slot_ids) && slot_ids[0] == slot.id()) {
                const auto* pred = down_cast<const VectorizedInConstPredicate<SlotType>*>(root_expr);

                if (!pred->is_join_runtime_filter()) {
                    continue;
                }

                // Ensure we don't compute this conjuncts again in olap scanner
                normalized_conjuncts[i] = true;

                if (pred->is_not_in() || pred->null_in_set() ||
                    pred->hash_set().size() > config::max_pushdown_conditions_per_column) {
                    continue;
                }

                std::set<RangeValueType> values;
                for (const auto& value : pred->hash_set()) {
                    values.insert(value);
                }
                range->add_fixed_values(FILTER_IN, values);
            }
        }
    }

    // bloom runtime filter
    for (const auto it : runtime_filters->descriptors()) {
        const RuntimeFilterProbeDescriptor* desc = it.second;
        const JoinRuntimeFilter* rf = desc->runtime_filter();
```

```
        using ValueType = typename vectorized::RunTimeTypeTraits<SlotType>::CppType;
        SlotId slot_id;

        // runtime filter existed and does not have null.
        if (rf == nullptr || rf->has_null()) continue;
        // probe expr is slot ref and slot id matches.
        if (!desc->is_probe_slot_ref(&slot_id) || slot_id != slot.id()) continue;

        const RuntimeBloomFilter<SlotType>* filter = down_cast<const RuntimeBloomFilter<SlotType>*>(rf);
        // For some cases such as in bucket shuffle, some hash join node may not have any input chunk from right table.
        // Runtime filter does not have any min/max values in this case.
        if (!filter->has_min_max()) continue;
        // If this column doesn't have other filter, we use join runtime filter
        // to fast comput row range in storage engine
        if (range->is_init_state()) {
            range->set_index_filter_only(true);
        }

        SQLFilterOp min_op = to_olap_filter_type(TExprOpcode::GE, false);
        ValueType min_value = filter->min_value();
        range->add_range(min_op, static_cast<RangeValueType>(min_value));

        SQLFilterOp max_op = to_olap_filter_type(TExprOpcode::LE, false);
        ValueType max_value = filter->max_value();
        range->add_range(max_op, static_cast<RangeValueType>(max_value));
    }
}
```